

Gadgets: Lazy Functional Components for Graphical User Interfaces

Rob Noble and Colin Runciman
email: {rjn,colin}@minster.york.ac.uk

Dept. of Computer Science, University of York, UK

Abstract. We describe a process extension to a lazy functional programming system, intended for applications with graphical user interfaces (GUIs). In the extended language, dynamically-created processes communicate by asynchronous message passing. We illustrate the use of the language, including as an extended example a simple board game in which squares are implemented as concurrent processes. We also describe a window manager, itself implemented in the extended functional language.

Keywords: functional language, processes, concurrency, window manager, Gofer.

1 Introduction and Motivation

Most of the time, elements of a graphical user interface (GUI) operate independently. For example, a menu doesn't interact with the rest of the program until the user selects an option. The user can highlight options, open up further menus or move the menu around the screen, all without doing anything that should concern any other element of the program. Popular languages such as C do not readily lend themselves to the job of programming interfaces like these. Programs end up contorted by event-polling loops or a multitude of callback functions.

The Fudgets [CH93] system has shown that lazy functional languages have advantages over imperative languages when it comes to writing programs with GUIs. Fudgets interact with each other by message passing. A Fudget may have a representation on the screen, or may just work in the background (an *abstract* Fudget). Fudgets are linked or composed by *combinators* that also specify their relative layout. Each Fudget may have a private state associated with it. A library of useful Fudgets ranges from simple integer displayers to a complete text editor. It is simple to piece together these *building blocks* into a program with a graphical user-interface. Aside from the interface, the rest of the program may also take advantage of conceptual concurrency by being implemented across a number of abstract Fudgets.

However, each Fudget has only a single input and a single output through which to pass messages. Multiple channels of communication must be multiplexed explicitly. A Fudget has a pair of channels for I/O (eg. the screen representation) that are multiplexed automatically by the combinators. However,

the general connections of a program are more varied, and any change in the communication will need the multiplexing to be re-worked. A further effect of this restriction is that the layout combinators can only place together Fudgets that have a channel of communication between them.

1.1 Improving on Fudgets

We have extended a lazy functional language to support concurrent execution of component processes. Like Fudgets, these processes communicate through typed channels, but we have removed the restriction to single channels. In our *Gadget Gofer* system programs with a GUI may be written. Not only are application programs written in Gadget Gofer, but the window manager is too. We have developed a basic library of useful interface Gadgets (buttons, windows, etc.) and functions for glueing Gadgets together.

As a brief example of what can be done, Fig. 1 shows the program and screen image of an up-down bargraph display. It is simplified slightly; Sect. 4.3 gives a complete version. Both `button` and `bargraph` are defined in a Gadget library; these are *not* calls to an existing C library.

```

updown :: Gadget
updown =
  wire $ \u ->
  wire $ \d ->
  let b1 = button (\x->x+1) (op u) upArrow
      b2 = button (\x->x-1) (op d) downArrow
      bg = bargraph [ip u,ip d] in
  wrap (b1 <|> bg <|> b2)

```

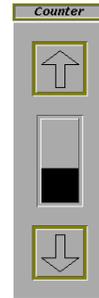


Fig. 1. An up-down bargraph display.

In Sect. 2 we describe our extended language. In Sect. 4 we see how we have used the language to implement programs with graphical user interfaces. Section 5 describes the screen manager. A longer example program is given in Sect. 6. Some issues of the implementation are discussed in Sect. 7. Section 8 briefly reviews related work. Section 9 concludes and suggests future work.

2 Components

Programs are networks of processes that communicate by message passing. Processes have input and output *pins* that are connected together with *wires* that convey the messages. This makes a diagram of the processes in a program look

like a circuit of electronic components, hence we call processes with pins *components*. A component may have any number of input and output pins. Message passing is asynchronous. Wires act like buffers, holding a queue of messages. Messages can be any first class value and are passed unevaluated. So functions, wires or even processes themselves may be passed as messages.

Predefined components that implement I/O devices provide a means to communicate with the user. For example, the *mouse* component has a single output pin that delivers *mouse-click* messages.

2.1 Component Definitions

The exact definition of primitive `Wire` and `Process` types is not important here. It is enough to know that a process is of type `Process s` where `s` is the type of a state value kept by the process. A wire of type `Wire a` will pass messages of type `a` only. Any attempt to send a message of the wrong type along a wire will result in a static type error. The pins of a component can be seen in its type. For example a component that has one input pin for messages of type `Int` and one output pin for messages of type `Char` has type `In Int -> Out Char -> Component`.

2.2 The Continuation Passing Style

One of the strengths of lazy functional languages is that the programmer can ordinarily abstract away from order of evaluation. In defining the operation of a component however, we need to state the order in which messages are sent and received. To do this we use the *continuation passing style* (CPS, described in [Kar81, HS88]). To avoid deeply nested brackets in continuations a right associative operator `$` for function application is useful: we write `f $ g $ h` instead of `f (g h)`.

2.3 Message Transmission

The function `tx` transmits a message from a component:

```
tx :: Out m -> m -> Process s -> Process s
```

Example: A simple component with one output pin sends the message “Hello World!” before ending using the primitive `end :: Process s`.

```
component :: Out String -> Component
component o = tx o "Hello World!" $ end
```

The style may look imperative and sequential. But there is no loss of referential transparency and it is only the parts in which the order of I/O is important that use continuation-based sequencing.

2.4 Message Reception

Because message passing is asynchronous and a component may have more than one input pin, we cannot tell where the next message will arrive from. A non-deterministic choice must be made when accepting the next message received. How can we introduce non-deterministic choice, whilst maintaining the referential transparency that is fundamental to a lazy functional language? We introduce a primitive that is like a restricted form of the `ALT` construct of Occam or `select` in PICT [PRT93]. The primitive is `from`, and is used *only* with another primitive, `rx`:

```
from :: In m -> (m -> Process s) -> Guarded s
rx  :: [Guarded s] -> Process s
```

The function `rx` takes a list of *guarded continuations* of the form:

```
from i $ c
```

where `i` is an input pin and `c :: m -> Process s` is the continuation that will be used if the next message is of type `m` and from pin `i`. Making guarded continuations an abstract type ensures that `from` cannot be used *outside* of an `rx` construct, and that only guarded continuations can be used *inside*. The `rx` primitive waits for a message to arrive on a pin, then picks the first guarded continuation that matches that pin. Referential transparency is maintained because the choice of pin to receive from is made outside the functional program, by the `rx` primitive.

Example: Figure 2 shows a component that implements a memory cell. A message received on one input pin causes the value stored to be emitted from the single output pin. A message on the other input pin replaces the stored value.

```
memory :: m -> In m -> In t -> Out m -> Component
memory s i t o =
  claim i $
  claim t $
  m s where
  m s =
    rx [
      from i $ \s' -> m s',
      from t $ \_ -> tx o s $ m s
    ]
```

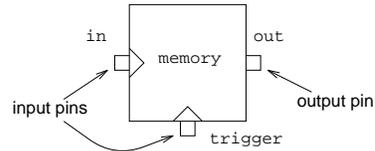


Fig. 2. A memory cell component.

The `claim` primitive is used by components to declare that they will receive messages from a particular wire. Components are sometimes passed wires

that they do not receive messages on, because they pass the wire on to another component. In this case, the component does not claim the wire. At the implementation level, a `claim` enables a simpler and more efficient scheduling algorithm to be used.

2.5 Composing Components

Components are often defined as compositions of other components. For this we use two more primitive functions:

```
wire :: (Wire a -> Process s) -> Process s
spawn :: Process a -> Process s -> Process s
```

where `wire` generates a new wire, passing it to its continuation argument, and `spawn` creates a new process out of the given process function. Generating a new wire or process requires a new value of the *world state* to be calculated, to which only processes have access. Therefore a composition of processes must itself be created by a process. Often this composing process exists only long enough to wire together and “switch on” the constituents of a composition. The assembled components remain separate as far as process scheduling is concerned, but appear from the outside to be a single component.

Example: Two memory components can be composed to form a memory storing two values, both released by the same trigger. Figure 3 shows the definition and wiring diagram. The principal tools used here are the primitives `wire` (to obtain a new wire) and `spawn` (to create a new process). A wire has two ends. Applying the function `ip` (or `op`) to a wire yields the end attached to an input (or output) *pin*.

```
two_memory :: m -> n -> In m -> In n ->
  In t -> Out m -> Out n -> Component
two_memory s1 s2 i1 i2 t o1 o2 =
  wire $ \a ->
  wire $ \b ->
  spawn (tee t (op a) (op b)) $
  spawn (memory s1 i1 (ip a) o1) $
  spawn (memory s2 i2 (ip b) o2) $
  end
```

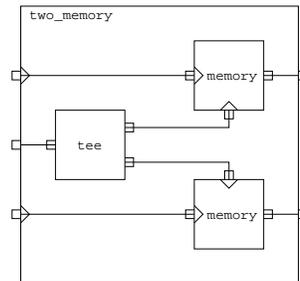


Fig. 3. A composition of three components.

2.6 Duplex Wires

Wires communicate messages in one direction only, but often two-way communication is required. It is convenient to define a type `Duplex a b` of paired wires that communicate messages of type `a` in one direction and of type `b` in the other.

```
type InOut a b = (In a, Out b)
type Duplex a b = (InOut a b, InOut b a)
```

The `duplex` function supplies a new duplex wire and is defined in terms of `wire`:

```
duplex :: (Duplex a b -> Process s) -> Process s
duplex c =
  wire $ \x ->
  wire $ \y ->
  c ((ip x,op y),(ip y,op x))
```

2.7 Some Useful Higher-Order Functions

We call a function of type `Process s -> Process s` an *action* because it describes one action in the operation of a process. When combined with a continuation, an action forms a complete process. Some actions pass on a value and have the type:

```
(r -> Process s) -> Process s
```

They must be applied to a continuation that expects a value of type `r`.

The higher-order functions `sequence` and `accumulate` are two of many useful *glue* functions:

```
sequence :: [Process s -> Process s] -> Process s -> Process s
accumulate :: [(r->Process s) -> Process s] ->
             ([r] -> Process s) -> Process s
```

Given a list of actions that *don't* pass on a value, and a continuation `c`, `sequence` performs the actions in the list one after another, then continues with `c`. Given a list of actions that *do* pass on a value, and a continuation `c`, `accumulate` performs the actions in the list one after another, storing each value passed on, then continues with `c` applied to the list of values collected.

2.8 Components With Lists of Pins

What if we want to define a component with several pins of the same type, but we won't know how many until the component is used in a program? A component can take a *list* of pins as argument. When receiving a message from one of a list of pins, the `froms` function is a useful alternative to `from`. Instead of a single input pin, `froms` expects a list of input pins each paired with some information that will be useful when responding to a message. The continuation given to `froms` expects a message (as with `from`) *plus* another value — the information that was paired with the corresponding input pin. For example, the `tag` component takes a list of input pins and a single output pin. It copies messages from any of the input pins to the output pin, pairing the message with an index number that indicates which pin the message arrived on.

```

tag :: [In m] -> Out (Int,m) -> Component
tag is o = sequence (map claim is) $ t
  where
    t = rx [froms (zip is [1..]) $ \m i -> tx o (i,m) $ t]

```

3 The Program Environment

All but the simplest of components are given a connection to a component called the *operating system* (OS) when they are created. Connections to resources such as a display screen are then obtained via the OS. A component with a connection to the OS is called a *System* component.

4 Gadgets

A Gadget is a component with an image on the screen. The image displays output from the Gadget. Mouse-clicks over the image generate input messages to the Gadget. Gadgets are hierarchical (the area occupied by a Gadget can contain further, smaller Gadgets) and may overlap on the screen (Gadget images are flat but may be placed at different depths). The screen hardware provides only a two-dimensional picture, gives no support for sharing the screen between multiple Gadgets, and cannot route user-input to the relevant Gadget. Instead, a *screen manager* component (SM) provides these facilities. It has one duplex connection to the *screen* and one duplex connection to each Gadget. When a Gadget sends a message to the SM, it can be thought of as communicating directly with its image on the screen.

Gadgets too can be defined directly or as a composition. In both cases, the wire connections to the SM are made implicitly and communication with the SM occurs without explicitly referring to these wires (using `txSM` and `fromSM` functions). This is possible because the SM wires are hidden within the state value held by the Gadget.

4.1 Gadget Compositions

As well as describing the connections between Gadgets, a Gadget composition must also define the relative layout of the Gadgets on the screen. This is normally achieved using *layout combinators* such as `<->` and `<|>`:

```

<->, <|> :: Gadget -> Gadget -> Gadget

```

that place Gadgets side-by-side (centred vertically) and one above the other (centred horizontally), respectively. Other combinators provide, for example, horizontal placement with Gadgets aligned at the top. Figure 4 shows a composition of a button and a bargraph. The composed Gadget operates like a bargraph, but has a button beside it that resets the bargraph to the lowest level. This is like a component composition, but instead of spawning the Gadgets to be combined, they are composed with the layout combinator `<->` and wrapped in a box large enough to hold the two.

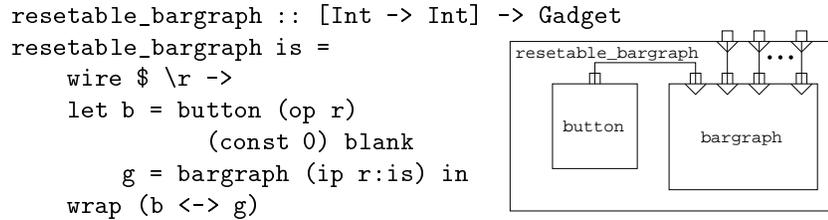


Fig. 4. A Resettable Bargraph.

4.2 Gadget Attributes

Gadgets have *attributes* such as width, height and colour. With many attributes to set, passing them all as arguments could lead to clutter and confusion. Even with only the attributes of width, height and colour a button application might be:

```
button (op w) Click 50 60 c
```

where “Click” is the message the button sends when clicked, “50” is the width, “60” is the height and “c” a colour. With only three attributes the button has five parameters, and it must be remembered which parameter controls each one.

Instead, suppose we define an algebraic type that specifies all a Gadget’s parameters, and assume that the Gadget has a *default value* of these *attributes*. We can pass the gadget a function to alter the default value. Take the `button` for example:

```
button :: (ButtonAttributes -> ButtonAttributes) ->
  Out m -> m -> Gadget
```

This is the *default Gadget*. It must be applied to some attribute-changing function before it becomes a Gadget. To leave the attributes at the default, we apply the default Gadget to the function `id`. Alternatively, *attribute modifying functions* may be composed and used to alter the attributes. For example, a button of default width and colour, but with a height of 100 and a *momentary* action (ie. the button pops out again when the mouse button is released):

```
button (height 100.buttonMomentary) (op w) Click
```

Some aspects of a Gadget’s operation are peculiar to the type of Gadget and have no sensible default value (eg. for a button, the output wire and message), so these values are separate parameters to the Gadget. Some Gadget attributes are specific to a particular Gadget (eg. `buttonMomentary`) but can have a default value. Others are common to many Gadgets (eg. `width` and `height`), and we would like to use the same name regardless of the particular type of Gadget. Here the *type classes* of Haskell prove useful [HHPJW94]. `Attributes` are made

a type class, of which an attribute type such as `ButtonAttributes` is an instance. Each type of `Gadget` defines its own overloaded versions of common modifiers such as `width` to alter its particular attribute datatype. Modifiers for attributes specific to a particular `Gadget` are not overloaded.

4.3 A Gadget Program

We are now in a position to look back at the updown bargraph display first introduced in Sect. 1.1 and see how it works. Figure 5 gives the complete version.

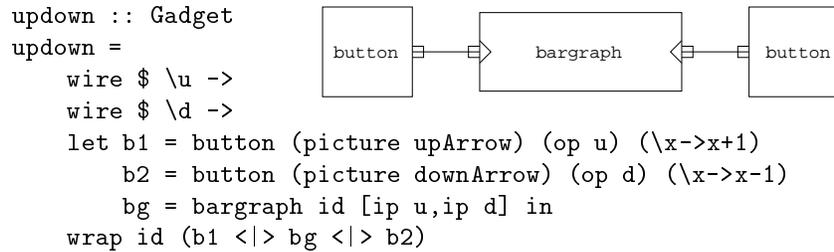


Fig. 5. The Wiring of updown.

Two wires, `u` and `d`, carry messages from each button to the bargraph display. The messages sent each time a button is clicked are *functions* of type `Int -> Int` that alter the current level of the bargraph. Altering the level with functions instead of simply stating a new level is more flexible and reduces the number of components required. It would be a simple matter to add a further button, for example, to reset the level to zero (as in Sect. 4.1).

The `<|>` combinator takes two `Gadgets` and places them one above the other. Layout components continue to exist throughout the life of the program, in case the `Gadgets` need to be re-positioned — eg. when a `Gadget` changes size. A layout component has no screen area of its own, but relies on a *container* supplied by `wrap` to hold the neighbouring `Gadgets`. By default, `wrap` supplies a blank box to contain its `Gadget` parameter, but the default can be changed with the `picture` attribute modifier, as used by the buttons.

5 The Screen Manager

Rather than provide an interface to an imperative window manager such as The X Windows System, we chose to program a screen manager in `Gadget Gofer`. Our motivation for doing this was to move the boundary between the functional and imperative worlds a step closer towards the hardware, increasing the applicability of functional languages, and to avoid the design constraints of

another programming paradigm (eg. event-polling loops or callbacks). Besides, others have already produced interfaces between lazy functional languages and imperative window managers (eg. [CH93, AvGP92, RS93, Sin92]).

Our aim is to hide the work involved in sharing the screen between layers of images (eg. redrawing images that become uncovered) from the Gadget programmer. A Gadget's connection to the SM may be viewed as a connection to the image itself. A new drawing sent to the image (really to the SM) changes the way it is displayed. Mouse-clicks or key-presses arrive (as if) from the image. We use the term *image* rather than *window* because these areas are used for a variety of interface items such as icon buttons, windows and window containers. The SM manages images in a hierarchy — a Gadget's image is created within the image of its parent. The Gadget end of the duplex wire connecting Gadget to SM is hidden within the Gadget state automatically when the Gadget is created.

Rather than sending actual drawing commands to the screen area, a Gadget sends a *drawing function* that the SM applies to *size* and *placement* values to generate a list of drawing commands. The benefit is that the SM can automatically redraw screen areas that become exposed without the need for further communication with relevant Gadgets.

```
type DrawFun = Size -> Placement -> [ScreenCmnd]
type Size = (Int,Int)
type Placement = (Coord,Coord)
type Coord = (Int,Int)
```

Examples of `ScreenCmnds` are `DrawLine (Coord,Coord)`, `DrawSetColour Colour`, and `DrawText Coord String`. The `Size` argument of a `DrawFun` indicates the size of the Gadget's screen area. The `Placement` argument indicates the position of a rectangle within the screen area that needs redrawing. When the SM uses a `DrawFun`, it only draws part of the area if another area is overlapping. The `DrawFun` can use the placement information to speed redrawing by filtering out commands that draw outside of the required region. If a `DrawFun` returns commands that draw outside of the required region they are clipped away.

Sending a new `DrawFun` results in the whole screen area of a Gadget being redrawn. In cases where only a small change is being made (eg. one extra line added) the Gadget may send a `DrawFun` that draws only what is being added.

The SM redraws the screen areas front-most first¹. A list of rectangles that need redrawing, `rs`, is maintained. Initially `rs` contains one rectangle the size of the whole screen. As each window is redrawn, the rectangle it occupies is subtracted from `rs`. The intersection of the position of a screen area with `rs` gives the placement argument for its `DrawFun`. The intersection may consist of more than one rectangle so the `DrawFun` may be called several times. But only the SM is involved, not the Gadget that supplied the `DrawFun`.

¹ The method is based on an unpublished algorithm due to Roger Took

5.1 Gadget Layout Management

Gadgets can specify their position relative to a corner of their parent's screen area, but they need not know about their on-screen neighbours. Decisions about their position are delegated to *layout components* (already described in Sect. 4.1). A parent Gadget specifies the layout of its children by wiring them to a layout manager using layout combinators. Gadgets with their respective layout managers are connected in an ancestral tree. At the top of the tree is the OS, as it owns the *root* window and is responsible for creating any program Gadgets the user wants to run.

When a program starts, layout managers collect a description of each image they are responsible for, package them together, and pass them up the tree. At the root of the tree this information is passed to the SM, where it forms the initial image data structure held by the SM. A Gadget may change the size of its image, sending the new size up the layout tree: this may affect the size of ancestors too (for example, if a Gadget is wrapped in a box that stretches to fit its contents), and several Gadgets may need to be repositioned. These changes are all handled transparently by the layout managers.

5.2 Gadget Connections

The SM, OS and layout wires of a Gadget are provided by its parent when a Gadget is created. They are kept in a hidden state and used by library functions such as `setSize` that sends new-size messages to the SM and the Gadget's layout manager.

In addition, a Gadget has connections specific to the job it performs. Typically, these pins are wired by the next level up the Gadget hierarchy. For example, in the updown bargraph display of Sect. 4.3, the wires `u` and `d` are connected by the parent, `updown`. Abstraction allows the programmer to forget about most of the wiring, concentrating on the main application wires only.

6 A Longer Example: Grid Explode!

6.1 The Game

The game of Grid Explode is played on a rectangular grid of cells. Each player has a supply of stones of a unique colour. Players take turns placing a stone into any cell that does not already contain another player's stones. The *neighbours* of a cell are the cells above, below, to the left and to the right of the cell. The *capacity* of a cell is the number of neighbours it has. If the total number of stones in a cell remains below the capacity after a player places a stone into it, then the turn is over and it is the next player's turn. When a cell reaches its capacity, it *explodes* invading each of its neighbours with one stone. A stone invading a cell turns any occupants to its colour, and may cause further explosions. The turn is over when all cells are below their capacity. A player wins if his or her move results in perpetual explosions or the explosions cease leaving every cell containing at least one stone of their colour.

6.2 The Gadget Version

The Gadget version assumes two players only. The picture on the right-hand side of Fig. 6 shows an example screen image. Each cell is a gadget, with a duplex connection for each of its four sides. This is the type of a *Grid Gadget* with four duplex wires, one for each side:

```
InOut m m -> InOut m m -> InOut m m -> InOut m m -> Gadget
```

A higher-order function, `grid`, wires grid Gadgets to their neighbours and places them in a grid on the screen:

```
grid :: [InOut m m] -> [InOut m m] -> [InOut m m] -> [InOut m m] ->
      [[InOut m m -> InOut m m -> InOut m m -> InOut m m -> Gadget]]
      -> Gadget
```

`grid` takes a list of duplex connections for each side of the grid and a list of rows of grid Gadgets. In this application, we do not use the edge connections in the grid, so give a list of (n_{ci}, n_{co}) values for each edge, where n_{ci} and n_{co} are special values of type `In a` and `Out a`, respectively, indicating that a pin is a *not-connected input* or *output*. In the main `explode` game function, we set the capacity of cells and wire them to a `referee` component, then pass all the cells to `grid`. The cells are wired as shown on the left-hand side of Fig. 6.

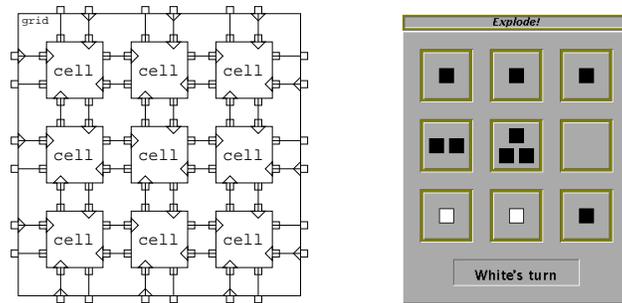


Fig. 6. The Programmer and Player Views of the Grid

To achieve a similar wiring in the Fudgets system would require a significant amount of multiplexing. Our initial design of the Explode game did not foresee the need for a referee component, so the links were added at a later stage. A new duplex pin was added to the cell and wired to the referee by a partial application of each cell function, before passing the resulting grid Gadgets to `grid`. To modify the wiring in this way with Fudgets would require the Fudget equivalent of `grid` to be completely re-written.

Stones are transmitted in messages to adjacent cells when an explosion occurs. Each cell knows when it has been clicked, how many stones it contains (if

any) and what colour they are, but knows nothing about the other cells in the grid. There are three problems to solve in this distributed implementation: (1) How does a cell determine what a click represents (ie. which player)? (2) How is termination detected? (3) How is the next player prevented from moving until the explosions from the previous move have completed?

To solve these problems each cell has a further duplex connection to a *referee* that negotiates the players turns and decides when a player has won. Cells send the referee messages of type `Notify`:

```
data Notify = ClickOver (Maybe Player)
            | Bonk (Maybe Player)
            | Boom Int (Maybe Player)
```

A `ClickOver` message indicates the colour of any stones already within the cell; a “`Boom n c`” message indicates an explosion, where `n` is the number of neighbouring cells exploded into and `c` is the colour of any stones previously in the cell; a cell that receives a stone, but does *not* explode, sends a “`Bonk c`” message, where `c` is the colour of any stones previously in the cell. The `cell` (Fig. 7) remembers its contents with a state value of type `Maybe (Player, Int)` where `Maybe a = Yes a | None` indicating that either the contents is empty with “`None`”, or that there are `n` stones belonging to player `p` with “`Yes (p,n)`”.

The referee (Fig. 8) sends cells `Rulings`:

```
data Ruling = Invasion Player | ClearSmoke
```

The referee remembers whose turn it is and when a cell sends a message saying it has been clicked and currently contains colour `c`, the referee checks that the move is valid, and if so returns an `Invasion` ruling, but otherwise makes no response. The referee is able to tell that explosions have ceased by keeping a tally. On receipt of a “`Boom n c`”, it increases the tally by `n-1`; On receipt of a “`Bonk c`”, it decreases the tally by one. When the tally reaches zero, the explosions have stopped, exploded cells are sent a `ClearSmoke` message and the next player may take a turn. To detect the end of the game the referee keeps a count of how many cells contain stones of each colour and which cells have exploded this move.

7 Implementation Outline

7.1 Communication Between Components

We have already seen that components are programmed using CPS. As well as enforcing an order on the operations of a component, the continuation functions also maintain a *state* for the process. Each continuation function, or *Action*, is able to modify this state — it is a state-transformer (ST). The state value consists of two or three parts:

1. the *world* of the component — a record of communications between this component and others. The world state is hidden in the definition of a component and not directly accessible. Special primitive ST’s describe message

```

cell :: Int -> InOut Ruling Notify ->
      Invade -> Invade -> Invade -> Invade -> Gadget
cell capacity (fromReferee,toReferee) l t r b =
  wire $ \d ->
  wire $ \c ->
  giveImage (button (pictureIn (ip d)) (op c) ()) $
  let draw = op d
      click = ip c
      cell' o =
        let neighbours = [l,t,r,b]
            invasion p o =
              let (op,n') = case o of
                  None -> (None,1)
                  Yes (p,n) -> (Yes p,n+1) in
              tx draw (stones p n') $
              if n' < capacity then
                tx toReferee (Bonk op) $
                cell' (Yes (p,n'))
              else
                tx draw boom $
                tx toReferee (Boom capacity op) $
                sequence [tx n p | (_,n) <- neighbours] $
                cell' None in
        rx [
          from click $ \_ ->
            let o' = case o of
                None -> None
                Yes (p,_) -> Yes p in
            tx toReferee (ClickOver o') $
            cell' o,
          from fromReferee $ \m -> case m of
            Invasion p -> invasion p o
            ClearSmoke -> tx draw blank $ cell' o,
          froms neighbours $ \p _ ->
            invasion p o
        ]
  in
  claim fromReferee $
  claim click $
  sequence (map (claim.fst) [l,t,r,b]) $
  cell' None

```

Fig. 7. The Operation of cell Defined.

```

referee :: [InOut Notify Ruling] -> Out Turn -> Component
referee cs turn =
  sequence (map (claim.fst) cs) $
  tx turn (Turn Black) $ p Black 0 everyone [] ih
  where
  p :: Player->Int->[CellID]->[CellID]->[(Player,Int)]->Component
  p c t b s h =
    rx [
      froms cs $ \r toCell ->
        let newcount h o n = map (adj (+1) n.adj (\x->x-1) o) h
            adj _ None h = h
            adj f (Yes np) (p,n) = (p,if p==np then f n else n)
        t' = t - 1 in
        case r of
        ClickOver v ->
          if (v == None || v == Yes c) && t == 0 then
            tx turn NoTurn $
            tx toCell (Invasion c) $
            p c 1 b s h
          else p c t b s h
        Bonk pc -> let h' = newcount h pc (Yes c)
                   s' = s \\ [toCell] in
          if t' == 0 then
            if (any (==nc) (map snd h')) then
              tx turn (Win c) $
              end
            else
              let c' = opponent c in
              tx turn (Turn c') $
              sequence [tx e ClearSmoke | e <- s'] $
              p c' 0 everyone [] h'
          else p c t' b s' h'
        Boom n pc -> let b' = b \\ [toCell] in
          if b' == [] then tx turn (Win c) $ end
          else p c (t'+n) b' (toCell:s) (newcount h pc None)
    ]
  everyone = map snd cs
  ih = zip players (repeat 0)
  nc = length cs

```

Fig. 8. The Operation of referee Defined.

transmission or reception by returning an altered world state. When a process is *executed* the changes in world state result in the I/O described being performed.

If we can ensure that only a single world state is in existence at any one time, then the primitive I/O functions can safely update the world value *in-place*. A similar optimisation is discussed in [P JW93].

2. the part used to record *default wire connections* in certain types of component (for example, the OS, SM and layout connections in a Gadget). These are hidden from component definitions but may be read or altered by the component. The type of this part of the state is the parameter to the `Process` type.
3. an optional part maintained at the component definition level and directly used by the component. For example, in the memory component of Sect. 2.4 an explicit state is used to hold the current memory contents.

7.2 Concurrency

Processes operate concurrently: evaluation is time-sliced between them. Each process is a separate evaluation with a private stack. Processes may share portions of the heap, and if two processes share an expression it is evaluated at most once. To ensure the integrity of the heap, a context-switch can only occur at certain points in the execution of a process, such as after a message has been sent.

7.3 Scheduling

When first created, a process is *initialising*. A process that has tried to receive when there are no messages waiting for it is *suspended*. The rest are *running*. Initialising processes are kept in a FIFO queue, and move to the running queue when they are about to receive or emit a message. Running processes are kept in a queue, ordered according to the number of messages waiting for them. Suspended processes are kept in a suspended list (in no particular order). Suspended processes that receive a message are moved to the running queue.

Whenever there is a change of context, the *scheduler* picks a process to be evaluated. Processes do not have a priority, but are scheduled according to the number of messages waiting for each process. The scheduler picks the top process from the initialising queue. If there are no processes initialising, then it picks the top process from the running queue. If there are no running processes, then no process is evaluated until a process becomes available on the running queue — for example, as a result of some user-input. After a running process has been evaluated to a point where a change of context can occur, it moves down the running queue to just below the lowest process with the same number of messages waiting. This has the effect that processes with the same number of messages waiting are chosen in a round-robin fashion.

This strategy was chosen to minimise the build-up of messages in wires. However, a situation could arise where a group of components starve those with fewer messages by communicating heavily with each other. In practice we have not seen this happen, but the risk could be reduced by including in the scheduling decision the length of time since a process last had a time-slot.

7.4 I/O Device Primitive Components

I/O devices such as the screen and mouse are implemented as primitive components. For each output pin of an I/O component (ie. input to the program), the wire-number² is noted so that when some input becomes available it is inserted into the relevant wire, ready to be received by the component at the other end. For the input pins of an I/O component (ie. output from the program), a C function is attached to the wire that will be called whenever a message arrives at the component. The message becomes an argument to the C function. Whenever the scheduler is called to make a change of context, it first calls a routine that polls to deliver any input (eg. mouse-click) to the relevant wire.

8 Related Work

There are many examples of work relating to message-passing concurrency in functional languages. One of the earliest is [Kar81]. One of the most recent is [JH93]. This system has a common ancestor with the system described in [WR95]. A few are concerned with the use of concurrency in programming applications with GUIs:

- Fudgets [CH93] behave in many respects like concurrent processes, but are actually implemented in a sequential language by exploiting the fact that Fudgets are restricted to one input and one output for messages, and must observe the *Fudget Law*. This makes composing and re-using Fudgets complicated. Fudgets are implemented in the lazy functional language Haskell. For a more detailed review, see [NR94].
- eXene [GR93] is an interface to The X Windows System, making full use of light-weight concurrent threads in the language (based on the strict functional language SML). It supports synchronous operations as first-class values, so new synchronisation abstractions can be built. The system is well developed and supports most of the functionality of The X Windows System.
- Erlang [AWV93] is a programming language for communicating processes, with an interface to The X Window System, used for programming real-time and control systems. The language borrows ideas from functional and concurrent languages, though it does not feature lazy evaluation or type-checking and is not a pure language.

² Internally, a wire is identified by an integer

- Pict [PRT93] is based on the π -calculus, so concurrency is central to the language. An interface to the The X Windows System is under development.
- Concurrent Clean [AP93] is a lazy functional language in which *unique types* are used to allow side-effecting I/O functions to operate safely. An interface to the The X Windows System is provided, using a system similar to callbacks and relying heavily on a global state for communication between elements of the program. More details can be found in [NR94].
- The Haggis system ([FJ95]) is based on a concurrent version of Haskell ([Jon95]) and features interface parts that are composable, extensible and concurrent. Communication between parts is achieved through synchronised access to shared memory.

Gadget Gofer is closest to the Fudgets system. It builds on the strengths of Fudgets by removing the restriction on channels per process: this simplifies the task of composing processes. It also untangles the interface layout from the communication structure of the application. The communication channels in Gadget Gofer are first class values, increasing the expressive power of the language.

9 Conclusions and Future Work

We have implemented an extension to a lazy functional language that gives us concurrent processes that communicate by passing messages via an unrestricted number of pins. This enables us to define independent components that are simple to compose, and program structures that are easy to change. Screen layout is independent of application communication structures, unlike the Fudgets system where layout and communication are closely tied. A screen manager built from the same components allows us to make good use of laziness and abstraction in defining interfaces. A small library of components and Gadgets, along with several example applications (such as the Explode game) have demonstrated the success of the system.

With the recent rise in popularity of the use of monads to encapsulate I/O (and more generally state-transformers) in lazy functional languages, why did we choose to use CPS? The reason is partly due to history — our system is based on some earlier work incorporating processes into a functional language that used CPS — and partly because the nature of continuation-based programs enables us to simplify the process implementation. However, we would like to benefit from the use of *lazy state threads* [LPJ94]. It is not yet clear whether this would require us to change from CPS to the monadic style of combinators and I/O, or whether we could implement lazy state threads under CPS.

There are shortcomings of the current implementation. For example, the creator of a component cannot destroy it at some later stage; each process can

only `end` its operation of its own accord. A component cannot select a subset of its input pins to receive the next message from, but must be ready to receive a message from any input at all times. We have found situations where it would be useful to wait for a message from a particular pin or subset of pins before receiving any messages from the rest. A revised `rx` construct could permit guards for only a subset of the input pins.

Generating user interfaces for applications is simplified a great deal by the use of higher-order functions provided in a library (like `sequence`) and written for particular applications (like `grid`), but the job of wiring together compositions still seems a little *low-level*. We plan to write an application (using Gadgets of course) to allow a programmer to piece together an interface of ready-built components graphically, connecting pins with wires by clicking on them, and placing Gadgets by dragging them across the screen. This should speed up the development of Gadget programs; it will also serve as a further application study.

The creation of wires and components in a composition is *dynamic*, but this is not immediately apparent because it is the first and only thing a composing process does. A more interesting case is where a component creates another component in response to receiving a message. We hope to explore this possibility more fully in due course.

10 Acknowledgements

We thank Malcolm Wallace, Roger Took, Sigbjørn Finne, the anonymous PLILP referees and others from the universities of York, Bristol, Glasgow and Kent, whose comments on this system have helped to shape its development.

References

- [AP93] P. Achten and R. Plasmeijer. The Beauty and the Beast, 93.
- [AvGP92] P. M. Achten, J. H. G. van Gronigen, and M. J. Plasmeijer. High level specification of I/O in functional languages. In *Fifth Annual Glasgow Workshop of Functional Programming, Ayr 6th-8th July 1992.*, July 1992.
- [AWV93] J. Armstrong, M. Williams, and R. Viriding. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.
- [CH93] M. Carlsson and T. Hallgren. Fudgets - a graphical user interface in a lazy functional language. In *Functional Programming and Computer Architectures*, pages 321–330. ACM Press, June 1993.
- [FJ95] Sigbjørn Finne and Simon Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics*, Maastricht, Netherlands, September 1995.
- [GR93] Emden R. Gansner and John H. Reppy. A Multithreaded Higher-order User Interface Toolkit. *User Interface Software: Software Trends*, 1:61–80, 1993.

- [HHPJW94] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in haskell. In *European Symposium on Programming*, volume 788 of *LNCS*. Springer-Verlag, April 94.
- [HS88] P. Hudak and R. S. Sundaresh. On the expressiveness of purely functional I/O systems. Technical report, Yale University Research Report YALEU/DCS/RR-665, Dept. of Computer Science, December 1988.
- [JH93] Mark Jones and Paul Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, Aug 93.
- [Jon95] Simon Peyton Jones. Concurrent haskell. In *Haskell Workshop*, La Jolla, June 1995.
- [Kar81] K. Karlsson. Nebula: A functional operating system. Technical report, Laboratory for Programming Methodology, Chalmers University of Technology and University of Goteb, 1981.
- [LPJ94] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.
- [NR94] Rob Noble and Colin Runciman. Functional languages and graphical user interfaces — a review and a case study. Technical Report YCS-94-223, Department of Computer Science, University of York, 1994.
- [PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages*, Jan 93.
- [PRT93] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.
- [RS93] A. Reid and S. Singh. Implementing Fudgets with standard widget sets. In *Glasgow functional programming workshop*, pages 222–235. Springer-Verlag, 93.
- [Sin92] Duncan C. Sinclair. Graphical user interfaces for Haskell. In J. Launchbury and Patrick M. Samson, editors, *Glasgow functional programming workshop*. Springer-Verlag, 1992.
- [WR95] Malcolm Wallace and Colin Runciman. Extending a functional programming system for embedded applications. *Software Practice & Experience*, 25(1):73–96, January 1995.